CHAPTER 19

Debugging Java in the Database

About this chapter

This chapter describes how to use the Sybase Java debugger to assist in developing Java in the database.

Contents

Introduction to debugging Java

With Java in the database, you can add complex classes into your database. In order to test these classes and to fix problems with them, you can use the Sybase Java debugger.

This chapter describes how to set up and use the Java debugger.

How the debugger works

The Java debugger is a Java application that runs on a client machine. It connects to the database using the Sybase jConnect JDBC driver.

The debugger debugs classes running in the database. You can step through the source code for the files as long as you have the Java source code on the disk of your client machine. (Remember, the compiled classes are installed into the database, but the source code is not).

Requirements for using the Java debugger

You need the following in order to use the Java debugger:

- **A Java runtime environment** The Java debugger is a client application running on your machine. You need a Java runtime environment such as the Sun Microsystems Java Runtime Environment or the full Sun Microsystems JDK on your machine in order to run the debugger.
- **Sybase jConnect** The debugger connects to the database using the Sybase jConnect JDBC driver. It requires features supported by jConnect that may not be supported by other JDBC drivers. Sybase jConnect is included with Adaptive Server Anywhere.
- **Source code** You need the source code for your application on your client machine.
- **Permissions** In order to use the debugger, you must either have DBA authority or be granted permissions in the SA_DEBUG group. This group is added to all databases when the database is created.

What you can do with the debugger

You can carry out many tasks with the Sybase Java debugger, including the following:

- **Trace execution** Step line by line through the code of a class running in the database. You can also look up and down the stack of functions that have been called.
- **Set breakpoints** Run the code until you hit a breakpoint, and stop at that point in the code.
- **Set break conditions** Breakpoints include lines of code, but you can also specify conditions when the code is to break. For example, you can stop at a line the tenth time it is executed, or only if a variable has a particular value. You can also stop whenever a particular exception is thrown in the Java application.
- **Browse classes** You can browse through the classes installed into the database.
- **Inspect and set variables** You can inspect the values of variables alter their value when the execution is stopped at a breakpoint.
- **Inspect and break on expressions** You can inspect the value of a wide variety of expressions.

A debugging tutorial

This section takes you through a simple debugging session.

Prepare the database

To prepare the sample database for this tutorial, you should run the script *jdemo.sql*, which installs the Java examples into the database. The class we use in this tutorial is the **`JDBCExamples`** class.

For instructions on how to install the Java examples, see "Installing the Java examples" on page 466.

For a discussion of the **`JDBCExamples`** class and its methods, see "Data Access Using JDBC" on page 503.

Prepare to run the Java debugger

Before you can run the debugger, you should ensure that your CLASSPATH environment variable can locate the classes it requires.

❖ **To set your CLASSPATH environment variable:**

1. The Sybase Java debugger is the file *Debug.jar*, installed in the *Java* subdirectory of your Adaptive Server Anywhere installation directory. If it is not already present, add this file to your CLASSPATH environment variable.
2. The debugger uses the Sybase jConnect JDBC driver to connect to the database. Sybase jConnect is the file *jdbcdrv.zip* in the *Java* subdirectory of your Adaptive Server Anywhere installation directory. If it is not already present, add this file also to your CLASSPATH environment variable.

Your CLASSPATH environment variable may look as follows after adding these files:

```
path\Java\Debug.jar;path\Java\jdbcdrv.zip;\jdk1.1.3\lib\
classes.zip
```

where *path* is your Adaptive Server Anywhere installation directory.

Now you are ready to start the debugger.

Start the Java debugger

The debugger runs on your client machine. As it is a Java application itself, it is run using your Java Virtual machine.

You can start the Java debugger from the command line or from Sybase Central.

❖ **To start the Java debugger from Sybase Central:**

1. Start Sybase Central (Windows Edition) and open the Utilities folder, under Adaptive Server Anywhere.
2. Double-click the Java debugger icon in the right panel.

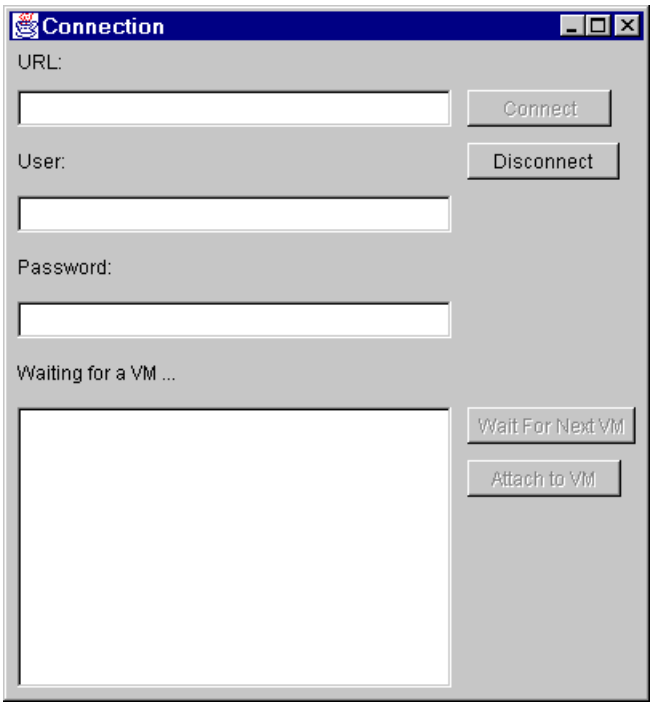
❖ **To start the Java debugger from a system prompt:**

1. From a system command prompt, change directory to your Adaptive Server Anywhere installation directory.
2. Enter the following command to run the Java VM (*java.exe*) using the debugger class:

   ```
   java sybase.vm.Debug
   ```

   The **`sybase.vm.Debug`** class is held in the *Debug.jar* file.

When you have started the Java debugger, the Connection window appears:



You can now connect to a database from the debugger.

Connect to the sample database

To connect to a database, you need to supply a URL, a user ID, and a password. This section describes by example how to connect.

❖ **To connect to the sample database from the debugger:**

1. Start a personal database server on the sample database. You can do this from the Start menu or by entering the following command at a system prompt:

   ```
   dbeng6 -c 8M path\asademo.db
   ```

   where *path* is your Adaptive Server Anywhere installation directory.

2. In the Java debugger, enter the following URL:

   ```
   jdbc:sybase:Tds:localhost:2638
   ```

   The meaningsof the components of this URL are described in "Supplying a URL for the server" on page 525.

3 Enter the user ID **DBA** and the password **SQL** and click Connect to connect to the database.

Once the connection is established, the debugger window displays the message Waiting for a VM. This is because no VM is currently running in the database under the **dba** user ID. If the **dba** user ID had one or more current connections carrying out Java operations, one VM for each connection would be listed in the VM list box.

Attach to a VM

Here, we start a VM for the **dba** user ID. You cannot start Java execution from the debugger. To start a VM you must carry out a Java operation from another connection under the same user ID.

❖ **To attach to a VM:**

1 With the debugger running, connect to the sample database from Interactive SQL as user ID **DBA**.

2 Execute some Java code using the following statement:

```
SELECT JDBCExamples.Query()
```

The Sybase Java VM starts in order to retrieve the Java objects from the table. The debugger immediately stops execution of the Java code. You will see that the command in ISQL does not complete (the EXECUTE button is grayed out, and no result set is displayed).

The debugger Connection window lists the VM in its list of available VMs.

3 In the debugger Connection window, click the VM and click ATTACH TO VM. The debugger attaches to the VM and the Source window appears. At this point the Connection window disappears.

The Source window is empty. The next step is to enable the Source window to show the source code for the method. The source code is available on disk.

Load source code into the debugger

The debugger looks in a set of locations for source code files (with *.java* extension). You need to add the *jxmp* subdirectory of your installation directory to the list of locations, so that the code for the class currently being executed in the database is available to the debugger.

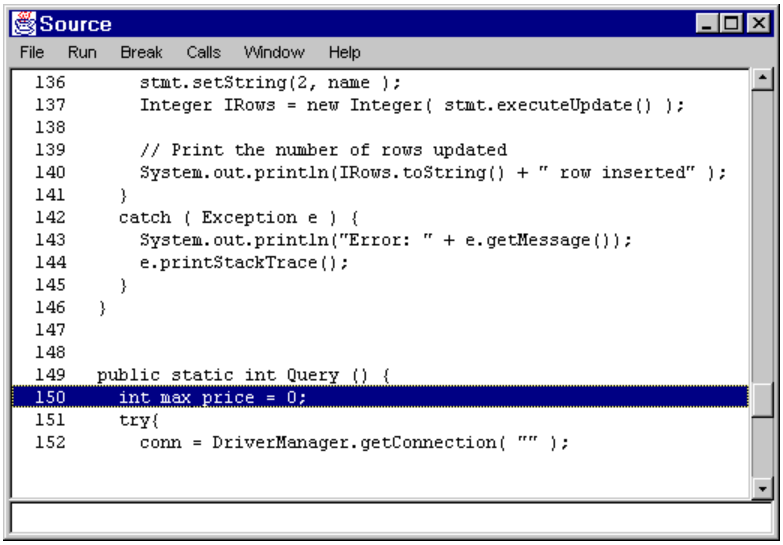
❖ **To add a source code location to the debugger:**

1. From the Source window, select File➤Source Path. The Source Path window displays.
2. From the Source Path window, select Path➤Add. Enter the following location into the text box:

   `path\jxmp`

   where *path* is the name of your Adaptive Server Anywhere installation directory.

   The source code for the **`JDBCExamples`** class displays in the window, with the first line of the Query method highlighted. The Java debugger has stopped execution of the code at this point.



   You can now close the Source Path window.

Step through source code

You can step through source code in the Java debugger in several ways. In this section we illustrate the different ways you can step through code using the **`Query`** method.

When execution pauses at a line until you provide further instructions, we say that the execution **breaks** at the line. The line is a **breakpoint**. Stepping through code is a matter of setting explicit or implicit breakpoints in the code, and executing code to that breakpoint.

Following the previous section, the debugger should have stopped execution of **`JDBCExamples.Query`** at the first statement:

Examples

Here are some example steps you can try:

1 **Step to the next line** Press F7 to step to the next line in the current method. Try this two or three times.

2 **Run to a selected line** Select the following line (line 284) using the mouse, and press F6 to run to that line and break:

```
max_price = price;
```

3 **Set a breakpoint and execute to it** Select the following line (line 292) and press F9 to set a breakpoint on that line:

```
return max_price;
```

An asterisk appears in the left hand column to mark the breakpoint. Press F5 to execute to that breakpoint.

4 **Experiment** Try different methods of stepping through the code. End with F5 to complete the execution.

When you have completed the execution, the Interactive SQL Data window displays the value 24.

Options

The complete set of options for stepping through source code are displayed on the ***Run*** menu. They include the following:

Function	Shortcut key	Description
Run	F5	Continue running until the next breakpoint, until the Stop item is selected, or until execution finishes.
Step Over	F7 or SPACE	Step to the next line in the current method. If the line steps into a different method, step over the method, not into it. Also, step over any breakpoints within methods that are stepped over.
Step Into	F8 or i	Step to the next line of code. If the line steps into a different method, step into the method.
Step Out	F11	Complete the current method, and break at the next line of the calling method.
Stop		Break execution.
Run to Selected	F6	Run until the currently selected line is executed and then break.
Home	F4	Select the line where the execution is broken.

Inspecting and modifying variables

You can inspect the values of both local variables (declared in a method) and class static variables in the debugger.

Inspecting local variables

You can inspect the values of local variables in a method as you step through the code, to better understand what is happening. You must have compiled the class with the *javac* `-g` option to do this.

❖ **To inspect and change the value of a variable:**

1. Set a breakpoint at the first line of the **`JDBCExamples.Query`** method. This line is as follows:

   ```
   int max_price = 0
   ```

2. In Interactive SQL, enter the following statement again to execute the method:

   ```
   SELECT JDBCExamples.Query()
   ```

   The query executes only as far as the breakpoint.

3. Press F7 to step to the next line. The **`max_price`** variable has now been declared and initialized to zero.

4. From the Source window, select Window➤Locals. The Local window appears.

   The Locals window shows that there are several local variables. The **`max_price`** variable has a value of zero. All others are listed as not in scope, which means they are not yet initialized.

5. In the Source window, press F7 repeatedly to step through the code. As you do so, the values of the variables appear in the Locals window.

   If a local variable is not a simple integer or other quantity, then as soon as it is set a + sign appears next to it. This means the local variable has fields that have values. You can expand a local variable by double-clicking the + sign or setting the cursor on the line and pressing ENTER.

6. Complete the execution of the query to finish this exercise.

Modifying local variables

You can also modify values of variables from the Locals window.

❖ **To modify a local variable:**

1. In the debugger Source window, set a breakpoint at the following line in the **`Query`** method of the **`JDBCExamples`** class:

   ```
   int max_price = 0
   ```

2. Step past this line in the execution.

3 Open the Locals window. Select the **`max_price`** variable, and select Local➤Modify. Alternatively, you can set the cursor on the line and press ENTER.

4 Enter a value of 45 in the text box, and click OK to confirm the new value. The **`max_price`** variable is set to 45 in the Locals window.

5 From the Source window, press F5 to complete execution of the query. In the Interactive SQL Data window, the value 45 is returned from the function.

Inspecting static variables

You can also inspect the values of class-level variables (static variables).

❖ **To inspect a static variable:**

1 From the debugger Source window, select Window➤Classes. The Classes window is displayed.

2 Select a class in the left hand box. The methods and static variables of the class are displayed in the right hand boxes.

3 Select Static➤Inspect. The Inspect window is displayed. It lists the variables available for inspection.

Using the debugger

This section describes how to accomplish tasks using the Java debugger.

Starting the debugger

The debugger is the Jar file *Debug.jar*, installed into your Adaptive Server Anywhere installation directory. This jar file contains many classes. To start the debugger you invoke the **`sybase.vm.Debug`** class, which has a **`main`** method.

❖ **To start the debugger:**

1. Ensure that *Debug.jar* and *jdbcdrv.zip* are in your CLASSPATH environment variable.
2. Enter the following command at the command line:

```
java sybase.vm.Debug
```

Connecting to a database on startup

When you start the debugger in the simple manner described above, you need to provide entries into the text boxes in order to connect to a database. You can also supply additional command-line arguments to the debugger to connect on startup.

You can specify these connection parameters in the following ways:

- **Connection string format** You can provide a `-c` command-line switch followed by a connection string consisting of the parameters URL, UID and PWD:

  ```
  java sybase.vm.Debug -c "uid=dba;pwd=sql"
  ```

  The UID and PWD connection parameters represent the user ID and password, respectively. The URL in this connection string may make use of the following default behavior:

  - **Full URL** You can provide a full URL of the form **jdbc:sybase:Tds:machine-name:port**.
  - **machine-name:port** You can omit the **jdbc:sybase:Tds** portion of the URL.
  - **Default port** If you do not specify a port number, **2638** is used as the default.
  - **Default machine name** If you do not specify a machine name, **localhost** is used as the default.

- **Default URL** You can omit the URL entirely, and the above defaults are used to construct a default URL.

- **Individual parameters** You can provide the following parameters, in order:
  - url
  - user ID
  - password

  For example, the following command (entered all on one line) connects to a server on the current machine, using user ID **DBA** and password **SQL**:

  ```
  java sybase.vm.Debug jdbc:sybase:Tds:localhost:2638
  DBA SQL
  ```

Compiling classes for debugging

Java compilers such as the Sun Microsystems *javac* compiler can compile Java classes at different levels of optimization. You can opt to compile Java code so that information used by debuggers is retained in the compiled class files.

If you compile your source code without using switches for debugging, you can still step through code and use breakpoints. However, you cannot inspect the values of local variables.

To compile classes for debugging using the *javac* compiler, use the `-g` command-line option:

```
javac -g ClassName.java
```

Attaching to a VM

When you connect to a database from the debugger, the Connection window shows all currently active VMs under the user ID. If there are none, the debugger goes into **wait mode**. Wait mode works like this:

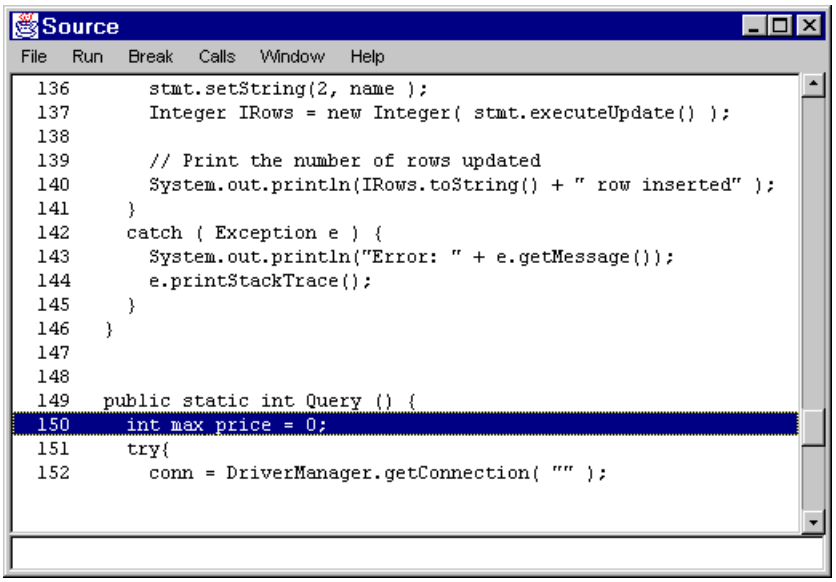
- Each time a new VM is started, it shows up in the list.
- You may either choose to debug a VM, or wait for another one to appear.

- Once you have passed on a VM, you lose your chance to debug that VM. If you then decide to attach to the VM, you must disconnect from the database and reconnect. At this time, the VM appears as a currently active VM and you can attach.

The Source window

The Source window serves the following purposes:

- It displays Java source code, with line numbers and breakpoint indicators (an asterisk in the left column).
- It displays execution status in the status box at the bottom of the window.
- It provides access to other debugger windows from the menu.



The debugger windows

The debugger has the following windows:

- **Breakpoints window** Displays the list of current breakpoints.
- **Calls window** Displays the current call stack.

- **Classes window** Displays a list of classes currently loaded in the VM. In addition, this window displays a list of methods for the currently selected class and a list of static variables for the currently selected class. In this window you can set breakpoints on entry to a method or when a static variable is written.
- **Connection window** The Connection window is shown when the debugger is started. You can display it again if you wish to disconnect from the database.
- **Exceptions window** You can set a particular exception on which to break, or choose to break on all exceptions.
- **Inspection window** Displays current static variables, and allows you to modify them. You can also inspect the value of a Java expression, such as the following:
  - Local variables
  - Static variables
  - Expressions using the dot operator
  - Expressions using subscripts []
  - Expressions using parentheses, arithmetic, or logical operators.

  For example, the following expressions could be used:

  ```
  x[i].field
  q + 1
  i == 7
  (i + 1)*3
  ```
- **Locals window** Displays current local variables, and allows you to modify them.
- **Status window** Displays messages describing the execution state of the VM.

Setting breakpoints

When you set a breakpoint in the debugger, the VM stops execution at that breakpoint. Once execution is stopped, you can inspect and modify the values of variables and other expressions in order to better understand the state of the program. You can then trace through execution step by step to identify problems.

Setting breakpoints in the proper places is a key to efficiently pinpointing the problem execution steps.

The Java debugger allows you to set breakpoints not only on a line of code, but on many other conditions. This section describes how to set breakpoints using different conditions.

Breaking on a line number

When you break on a particular line of code, execution stops whenever that line of code is executed.

❖ **To set a breakpoint on a particular line:**

♦ In the Source window, select the line and press F9.

Alternatively, you can double-click a line.

When a breakpoint is set on a line number, the breakpoint is shown in the Source window by an asterisk in the left hand column. If the Breakpoints window is open, the method and line number is displayed in the list of breakpoints.

You can toggle the breakpoint on and off by repeatedly double-clicking or pressing F9.

Breaking on a class method

When you break on a method, the break point is set on the first line of code in the method that contains an executable statement.

❖ **To set a breakpoint on a class method:**

1 From the Source window, choose Break➤ New. The Break At window is displayed.

2 Enter the name of a method in which you wish execution to stop. For example

JDBCExamples.Query

stops execution whenever the `JDBCExamples.Query` method is entered.

When a breakpoint is set on a method, the breakpoint is shown in the Source window by an asterisk in the left hand column of the line where the breakpoint actually occurs. If the Breakpoints window is open, the method is displayed in the list of breakpoints.

Using counts with breakpoints

If you set a breakpoint on a line that is in a loop, or in a method that is frequently invoked, you may find that the line is executed many times before the condition you are really interested in takes place. The debugger allows you to associate a count with a breakpoint, so that execution stops only when the line is executed a set number of times.

❖ **To associate a count with a breakpoint:**

1. From the Source window, select Break➤Display. The Breakpoints window is displayed.
2. In the Breakpoints window, click a breakpoint to select it.
3. Select Break➤Count. A window is displayed with a field for entering a number of iterations. Enter an integer value. The execution will stop when the line has been executed the specified number of times.

Example

This example assumes you have run the tutorial described in "A debugging tutorial" on page 536.

The **`JDBCExamples.Query`** method used in the tutorial has a loop in it:

```
while( result.next() ) {
   int price = result.getInt(2);
   if( price > max_price ) {
     max_price = price ;
   }
```

The lines inside this loop are executed ten times. You can set a breakpoint on the **`if`** statement and associate a count of two with it. Then execute the query to stop at the breakpoint.

Count is decremented

The count is decremented each time the line of code is executed. A side effect of this is that once the breakpoint is reached, the count is set to zero.

Using conditions with breakpoints

The debugger allows you to associate a condition with a breakpoint, so that execution stops only when the line is executed and the condition is met.

❖ **To associate a condition with a breakpoint:**

1. From the Source window, select Break➤Display. The Breakpoints window is displayed.
2. In the Breakpoints window, click a breakpoint to select it.

3 Select Break➤Condition. A window is displayed with a field for entering an expression. The execution will stop when the condition is true.

The expressions used here are the same as those that can be used in the Inspection window, and include the following:

- Local variables
- Static variables
- Expressions using the dot operator
- Expressions using subscripts []
- Expressions using parentheses, arithmetic, or logical operators.

Example

This example assumes you have run the tutorial described in "A debugging tutorial" on page 536.

The **`JDBCExamples.Query`** method used in the tutorial has a loop in it:

```
while( result.next() ) {
   int price = result.getInt(2);
   if( price > max_price ) {
     max_price = price ;
   }
```

The lines inside this loop are executed ten times. You can set a breakpoint on the **`if`** statement and associate the following condition with it:

```
price > 10
```

Then execute the query to stop at the breakpoint whenever the price is greater than $10.

Breaking when execution is not interrupted

With a single exception, breakpoints can only be set when program execution is interrupted. If you clear all breakpoints, and run the program you are debugging to completion, you can no longer set a breakpoint on a line or at the start of a method. Also, if a program is running in a loop, execution is continuing and is not interrupted.

To debug your program under either of these conditions, select Run➤Stop from the Source window. This stops execution at the next line of Java code that is executed. You can then set breakpoints at other points in the code.